The Euler Path to Static Level-Ancestors

Amir M. Ben-Amram
amirben@mta.ac.il

September 5, 2009

Abstract

Suppose that a rooted tree T is given for preprocessing. The *level-ancestor problem* is to answer quickly queries of the following form. Given a vertex v and an integer $i > 0$, find the ith vertex on the path from the root to v. Algorithms that achieve a linear time bound for preprocessing and a constant time bound for a query have been published by Dietz (1991), Alstrup and Holm (2000), and Bender and Farach (2002). The first two algorithms address dynamic versions of the problem; the last addresses the static version only and is the simplest so far. The purpose of this note is to expose another simple algorithm, derived from a complicated PRAM algorithm by Berkman and Vishkin (1990,1994). We further show some easy extensions of its functionality, adding queries for descendants and *level successors* as well as ancestors, extensions for which the formerly known algorithms are less suitable.

Keywords: algorithms, data structures, trees.

1 Introduction

The *level-ancestor* problem is defined as follows. Suppose that a rooted tree T is given for preprocessing. Answer quickly queries of the following form. Given a vertex v and an integer i, find an ancestor of v in T whose level is i, where the level of the root is 0.

Two related tree queries are: Level Successor—given v, find the next vertex (in preorder) on the same level. Level Descendant—given v and i, find the first descendant of v on level i (if one exists).

The level-ancestor problem is a relative of the better-known LCA (Least Common Ancestor) problem. In their seminal paper on LCA problems [12], Harel and Tarjan solve the level ancestor problem on certain special trees as a subroutine of an LCA algorithm. An application of the Level Ancestor problem is mentioned already in [1], although an implementation of this data structure had not yet been published at the time.

The first published algorithms for the level ancestor problem were a PRAM algorithm by Berkman and Vishkin [6, 7], and a serial (RAM) algorithm by Dietz [8] that accommodates dynamic updates. Alstrup and Holm [3] gave an algorithm that solves an extended dynamic problem, and has the additional advantage that its static-only

version is simpler than the previous algorithms. Finally, the simplest algorithm—for the static problem only—was given by Bender and Farach [5].

It is curious that very complicated algorithms to address theoretical challenges, namely dynamization and parallelization, had been published for this problem earlier than any simple algorithm for the most basic and useful variant (static, on serial RAM). It is also curious that the essential ideas for such an algorithm do appear in Berkman and Vishkin's solution but this potential contribution was missed, since they concentrated on the PRAM problem, for which they gave a notoriously impractical algorithm (involving a tableof almost $2^{2^{28}}$ entries). The first goal of this paper is to rectify this situation by presenting a sequential algorithm based on the approach of Berkman and Vishkin. This is not done just for historical interest, but because the algorithm here presented is simply useful: it is efficient and easy to implement (and has been implemented). Furthermore, we shall present a few useful extensions that were either unsupported by previous work, or supported in much more complicated ways. Specifically, we show how to accommodate level successor and level descendant queries, in addition to level ancestor. Together, these two queries are useful for iterating over the descendants of a vertex at a given level. For example applications of the extension, see [14, 15].

Technical remarks. Since we only consider data structures that support $O(1)$-time queries, we refer to the algorithms by the preprocessing cost. That is, an $O(n)$-time algorithm means linear-time preprocessing. The data to the algorithm is a tree T whose precise representation is of little consequence (since standard representations are interchangeable within linear time). We assume that vertices are identified by numbers 0 through $n-1$.

2 The Euler Tour and the Find-Smaller problem

Like the better-known LCA algorithm that also originates from [6], this Level Ancestor algorithm is based on the following key ideas:

- The *Euler Tour* representation of a tree reduces the problem to a problem on a linear array.
- A data structure with $\Theta(n \log n)$ preprocessing time (and size) is given for this problem.
- This solution is improved to linear-time preprocessing and size using the *microset technique* [10, 12].

The microset technique is also used in other work on level ancestors [3, 5, 13, 11] but they all apply at least part of the processing to the *tree*, using various methods of decomposition into subtrees. Here, all processing is applied to the Euler-tour array.

Consider a tree $T = (V, E)$, rooted at some vertex r. For each edge $(v \to u)$ in T, add its anti-parallel edge $(u \to v)$. This results in a directed graph H. Since the in-degree and out-degree of each vertex of H are the same, H has an Euler tour that

starts and ends in the root r of T. Note that the tour consists of $2(n-1)$ arcs, hence $2n-1$ vertices including the endpoints.

By a straight-forward application of DFS on T we can compute the following information:

1. An array $E[0..2n-2]$ such that $E[i]$ is the ith vertex on the Euler tour.
2. An array $L[0..2n-2]$ such that $L[i]$ is the level of the ith vertex on the Euler tour.
3. An array $R[0..n-1]$ such that $R[v]$ is the index of the last occurrence of v in the array E, called the *representative* of v.

Observation 1 *Let* $l < \text{level}(v)$. *Vertex* u *is the level-*l *ancestor of vertex* v *if and only if* u *is the first vertex after the last occurrence of* v *in the Euler tour such that* $\text{level}(u) \leq l$.

By this observation, the computation of the arrays E, L and R reduces the level-ancestor problem to the following

FIND-SMALLER (FS) Problem.

Input for preprocessing: Array $A = (a_1, a_2, \ldots, a_n)$ of integers

Query: Let $0 \leq i < n$ and $x \in \mathbb{Z}$. A query $\text{FS}_A(i, x)$ seeks the minimal $j > i$ such that $a_j \leq x$. If no such j exists, the answer is 0.

Our goal is to preprocess the array A so that each query $\text{FS}_A(i, x)$ can be processed in $O(1)$ time.

The Euler tour implies that the difference between successive elements of array L is exactly one. Therefore, for our goal, it suffices to solve the following restricted problem:

(± 1)FS is the Find-Smaller problem restricted to arrays A where for all i, $|a_i - a_{i+1}| = 1$.

We remark that the general Find Smaller problem cannot be solved with $O(1)$ query time, if one requires a polynomial-space data structure, and assumes a polylogarithmic word length; the reason is that the static predecessor problem, for which non-constant lower bounds are known [4], can be easily reduced to it.

Another preparatory definition is the following. Let n be a power of two and consider a balanced binary tree of $n-1$ nodes numbered 1 through $n-1$ in symmetric order (thus, 1 is the leftmost leaf and $n-1$ the rightmost). The height of node i is $\text{rnz}(i)$, the position of the rightmost non-zero bit in the binary representation of i, counting from 0. We denote by $\text{LCA}_{BT}(i, j)$ the least common ancestor of nodes i and j. For the algorithms, we assume that $\text{LCA}_{BT}(i, j)$ is computed in constant time. In fact, it can be computed using standard machine instructions and the MSB (most significant set bit) function; this function is implemented as an instruction in many processors, but could also be provided by a precomputed table. Following is a useful property of the rnz() function.

Lemma 2 *If $j < i$ are two nodes of the complete binary tree, and $k = \mathrm{LCA}_{BT}(j, i)$, then $i - j + 1 \leq 2^{1+\mathrm{rnz}(k)}$, and $i - k + 1 \leq 2^{\mathrm{rnz}(k)}$.*

We omit the easy proof. Finally, for uniformity of notation, we define $\mathrm{LCA}_{BT}(j, i)$ for $j \leq 0 < i$ to be 0.

3 Basic constant-time-query algorithm

In this section we describe an $O(n \log n)$-time preprocessing algorithm for the (± 1)FS problem. Throughout this section and the sequel, we make the simplifying assumption that n is a power of two.

Our description of the Basic algorithm has two steps. (1) The output of the preprocessing algorithm is specified, and it is shown how to process an FS query in constant time using this output. (2) The preprocessing algorithm is described. This order helps motivating the presentation.

3.1 Data structure and query processing

For each i, $0 \leq i < n$, the preprocessing algorithm constructs an array $B_i[1..f(i)]$, where $f(0) = n$ and for $i > 0$, $f(i) = 3 \cdot 2^{\mathrm{rnz}(i)}$. In $B_i[j]$ we store the answer to $\mathrm{FS}(i, a_i - j)$.

A query $\mathrm{FS}(i, x)$ is processed as follows (we assume that $x > a_0 - n$, for otherwise the answer is immediate, due to the ± 1 restriction).
(1) If $x \geq a_i$, return i.
(2) Let $d = a_i - x$. If $d \leq f(i)$ return $B_i[d]$.
(3) Otherwise, let $k = \mathrm{LCA}_{BT}(i - d + 1, i)$; return $B_k[a_k - x]$.

Figure 1 demonstrates the structure for a 16-element array A, except that all the arrays B_i are truncated to 8 elements. In this example, the query $\mathrm{FS}(6, 3)$ is answered immediately as $B_6[1] = 11$; the query $\mathrm{FS}(9, 1)$ is answered via Case (3): $k = 8$ and $B_k[a_k - 1] = B_8[5] = 13$.

We now explain the algorithm. Correctness of Case (2) is obvious by the definition of the structure. The correctness in Case (3) hinges on two claims. The first, Claim 3 below, shows that the reference to $B_k[a_k - x]$ is within bounds; the second, Claim 4, shows that the answer found there is the right one.

Claim 3 *In Case (3), we have $0 < a_k - x \leq f(k)$.*

Proof. For the first inequality: $k > i - d$ by its definition; we are dealing with (± 1)FS, therefore $a_k > a_i - d = x$. For the second inequality: We assume $k > 0$, as for $k = 0$ and the claim clearly holds. Consider the complete binary tree of $n - 1$ nodes, used to define $\mathrm{LCA}_{BT}(i, j)$. The algorithm sets $k = \mathrm{LCA}_{BT}(i - d + 1, i)$, so by Lemma 2,

$$
\begin{aligned}
2^{1+\mathrm{rnz}(k)} &> i - (i - d + 1) = d - 1 = a_i - x - 1 \\
2^{\mathrm{rnz}(k)} &> i - k \\
\Rightarrow \qquad 3 \cdot 2^{\mathrm{rnz}(k)} &\geq a_i - x + i - k.
\end{aligned}
$$

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a_i	0	1	2	3	2	3	4	5	6	5	4	3	2	1	2	1

j	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8	B_9	B_{10}	B_{11}	B_{12}	B_{13}	B_{14}	B_{15}
1	0	0	13	4	13	12	11	10	9	10	11	12	13	0	15	0
2	0	0	0	13	0	13	12	11	10	11	12	13	0	0	0	0
3	0	0	0	0	0	0	13	12	11	12	13	0	0	0	0	0
4	0		0		0		0		12		0		0		0	
5	0		0		0		0		13		0		0		0	
6	0		0		0		0		0		0		0		0	
7	0				0				0				0			
8	0				0				0				0			

Figure 1: The basic FS structure.

Since the difference between consecutive elements is ± 1, we have $a_k \le a_i + i - k$, so we conclude that

$$3 \cdot 2^{\mathrm{rnz}(k)} \ge a_k - x.$$

□

Claim 4 *If* $i - a_i + x < k \le i$, *then* $\mathrm{FS}(k,x) = \mathrm{FS}(i,x)$.

Proof. Because we are dealing with (± 1)FS, the values $a_k, \ldots, a_i$ are all in the interval $(a_i - (i-k), a_i + (i-k))$. By assumption we have $a_i - (i-k) > x$. Thus, the answer to $\mathrm{FS}(k,x)$ lies beyond a_i, and is also the answer to $\mathrm{FS}(i,x)$. □

3.2 The preprocessing algorithm

It is easy to verify that the size of the data structure is $\Theta(n \log n)$. To construct it in $O(n \log n)$ time, we perform a sweep from right to left; that is, for $i = n-1, n-2, \ldots, 0$ we compute an array $F[\min A, \ldots, \max A]$ where $F[x]$ is the index of the first $j > i$ such that $a_j = x$ (or 0 by default). Note that this is not the same as $\mathrm{FS}(i,x)$. Initializing F for $i = n-1$ is trivial and that updating it when i is decremented is constant-time. For each i, B_i is just a copy of an appropriate section of F. This completes the preprocessing.

4 Improved constant-time-query algorithm

In this section we describe an $O(n)$-time algorithm, based on the solution of the former section together with the *microset technique*. The essence of the technique is to fix a *block length* $b = \lfloor (\log n)/2 \rfloor$ and to sparsify the structure of the last section by using it only on block boundaries, reducing its cost to $O(n)$, while for *intra-block queries* we use an additional data structure, the *micro-structure*. For presentation's sake, we now provide a specification of the micro-structure and go on to describe the rest of the

structure. The implementation of the micro-structure will be dealt with in the following section.

For working with blocks, without resorting to numerous division operators, we shall write down some numbers (specifically, array indices) in a quotient-and-remainder notation, $ib + j$, where it is tacitly assumed that $0 \le j < b$.

The Micro-Structure. This data structure is assumed to support in $O(1)$ time the following query: Micro.FS$(ib + j, x)$—return the answer to FS$(ib + j, x)$ provided that it is less than $(i + 1)b$. Otherwise, return 0.

The FS Structure. For each i, $0 \le i < n/b$, our preprocessing algorithm now constructs two arrays:

1. A *near* array $N_i[1, \ldots, 2b]$ such that $N_i[j]$ stores the answer to FS$(ib, a_{ib} - j)$ (namely, the first $2b$ entries of B_{ib} of the previous section).

2. A *far* array $F_i[1, \ldots, f(i)]$ such that

$$F_i[j] = \lfloor \mathrm{FS}(ib, a_{ib} - jb)/b \rfloor$$

Thus, the arrays are not only sparsified, but also (for the far arrays) are their values truncated. Referring to the example in Figure 1, we have $b = 2$, so near arrays have 4 elements, e.g., $N_4 = (9, 10, 11, 12)$. The far array F_4 has $f(4) = 12$ entries: $(5, 6, 0, \ldots, 0)$.

The following fact follows from the (± 1) restriction and the definition of F_i:

Observation 5 *If $F_i[j] = k$, then $a_{ib} - jb \le a_{kb} < a_{ib} - (j - 1)b$.*

Query processing. A query FS$(ib + j, x)$ is processed as follows (we assume once again that $x > a_0 - n$).

(1) If $x \ge a_{ib+j}$, return $ib + j$.

(2) If $j = 0$ then

  (2.1) If $x \ge a_{ib} - 2b$ return $N_i[a_{ib} - x]$

  (2.2) $d \leftarrow \lfloor (a_{ib} - x)/b \rfloor$;
  if $d \le f(i)$ then

   (2.2.1) $k \leftarrow F_i[d]$; return FS(kb, x).
   else
   (2.2.2) $l \leftarrow i - d + 1$; $k \leftarrow \mathrm{LCA}_{BT}(l, i)$; return FS$(kb, x)$.

(3) (if $j > 0$)
 $m \leftarrow$ Micro.FS$(ib + j, x)$;
 if $m \ne 0$, return m, else return FS$((i + 1)b, x)$.

The following observations justify this procedure, and also show that there is no real recursion here: the recursive calls can actually be implemented as `goto`s and they never loop.

(1) In Case (3), when the micro-structure does not yield the answer, it follows that the element sought is further than $(i+1)b$; therefore the recursive call is correct, and will be handled at Case (2).

(2) In Case (2.2.1), we have (see Observation 5 above)

$$a_{kb} < a_{ib} - (d-1)b < x + 2b$$

and

$$a_{kb} \geq a_{ib} - db \geq x$$

Therefore, the recursive call is handled correctly at Case (2.1).

(3) For Case (2.2.2), we can show, as for the basic algorithm, that $\mathrm{FS}(kb, x) = \mathrm{FS}(ib, x)$ (same proof as before), and that $0 < a_{kb} - x \leq f(k) \cdot b$, showing that the recursive call falls back to Case (2.2.1). The last inequality is proved as Claim 6.

Claim 6 *In Case (2.2.2), we have* $a_{kb} - x \leq f(k) \cdot b$.

Proof. We assume $k > 0$. By Lemma 2,

$$\begin{aligned} 2^{1+\mathrm{rnz}(k)} &\geq i - l + 1 = d > \frac{a_{ib} - x}{b} - 1 \\ 2^{\mathrm{rnz}(k)} &\geq i - k + 1 \\ \Rightarrow \quad f(k) = 3 \cdot 2^{\mathrm{rnz}(k)} &> \frac{a_{ib} - x}{b} + i - k = \frac{a_{ib} - x + ib - kb}{b} \geq \frac{a_{kb} - x}{b}, \end{aligned}$$

where the last inequality is justified by the (± 1) property. □

5 The Micro Structure

The purpose of the micro structure is to support "close" queries, i.e., return the answer to $\mathrm{FS}(ib+j, x)$ provided that it is at most $(i+1)b$. There are several ways to implement this structure, with subtle differences in performance or ease of implementation. We describe two.

5.1 Berkman and Vishkin's structure

The basis for fast solution of in-block queries in [7] is observing that, up to normalization, there are less than 2^b different possible blocks. Normalization amounts to subtracting the first element of the block from all elements; i.e., moving the "origin" to zero. Clearly, a query on any array A, $\mathrm{FS}_A(j, x)$, is equivalent to $\mathrm{FS}_{A'}(j, x - a_0)$ where A' is the normalized form of A. The bound 2^b follows from the (± 1) restriction. This also allows us to conveniently represent a block as a binary string of length $b - 1$ (which fits in a word). We obtain the following solution.

Preprocessing: For every possible "small" array S of size b, beginning with 0, and satisfying the (± 1) restriction, build a matrix $M_S[b \times 2b]$ such that $M_S[j,x]$ is the answer to $\mathrm{FS}_S(j,x)$ for every $0 \le j < b$ and $-b < x < b$. As an identifier of S (to index the array of matrices) we use the $(b-1)$-bit representation of S. While preprocessing an array A of size n for FS queries, we store for every $0 \le i < n/b$ the identifier $S[i]$ of the block $(a_{ib}, \ldots, a_{(i+1)b-1})$.
Query: $\mathrm{Micro.FS}_A(ib+j,x)$ is answered by looking up $M_{S[i]}[j, x-a_{ib}]$ (returning 0 if the second index is out of range).
Complexity: The query is obviously constant-time. For the preprocessing, creating the idntifier array S clearly takes $\Theta(n)$ time. The construction of a single matrix M_S can be done quite simply in $\Theta(b^2)$ time, and altogether we get $2^b \cdot \Theta(b^2) = O(n)$ time and space.

5.2 A solution after Alstrup, Gavoille, Kaplan and Rauhe

Another implementation of the micro structure is suggested by an idea from [2]. In its basic form, as we next describe, it is really independent of the division into blocks—except that it only supports queries where the answer is close enough to the query index.

For $i < j \le n$, let

$$m(i,j) = \begin{cases} 1 & \text{if } a_j < \min\{a_i, \ldots, a_{j-1}\} \\ 0 & \text{otherwise.} \end{cases}$$

From the (± 1) property, one can easily deduce that $\mathrm{FS}_A(i, a_i - k)$ is precisely the position of the kth 1 in the sequence

$$m(i,i+1), m(i,i+2), \ldots$$

The solution to the micro-structure problem, based on this observation, follows:
Preprocessing: For every $0 \le i < n$, compute and store in an array entry $M[i]$ the b-bit mask $(m(i,i+1), \ldots, m(i,i+b))$.
Query: $\mathrm{Micro.FS}_A(i,x)$ is answered (for $x < a_i$) by looking up the $(a_i - x)$'th set bit in $M[i]$. The answer is 0 if there is no such bit.

This query returns answers in positions up to $i+b$, rather than $\lceil i/b \rceil \cdot b$, which can possibly result in a faster query. As an additional advantage, b can be enlarged up to the word size, saving both time and space (there is a certain caveat—see below).
Query Complexity: The query is constant-time if we have a constant-time implementation of the function that locates the i'th bit set in a word. In the absence of hardware support, a precomputed table, of size $O(2^b \cdot b)$, can be used (but this requires limiting the value of b as before)[1].
Preprocessing Algorithm: To compute the mask array M, we scan A from right to left while maintaining two pieces of data: the mask corresponding to the current position

[1] Another way, which is not constant-time in the RAM model, is to search for this bit using available arithmetic/logical instructions. Since this is a tight loop without any memory access, it may be even faster than a table access on a real computer.

i, and a stack that includes the indices $\{j \mid m(i,j) = 1\}$ up to the end of A. Each time the current position i is decremented, i is pushed unto the stack, possibly kicking off the top two elements (specifically, if $a_{i+1} = a_i + 1$). The current mask is easily adjusted in $O(1)$ time.

Clearly, the computation of M takes $\Theta(n)$ time, and this is also the space required. Fischer and Heun [9] propose to apply this technique within microblocks; in other words, revert to the Berkman-Vishkin approach of maintaining a table indexed by the block identifier, but keep the mask table instead of an explicit answer matrix. This saves a factor of b in the size of the micro structure, but is likely to be competitive in speed only if the bit-finding operation we make use of is supported by hardware.

5.3 Saving memory

In our description of the algorithm we aimed for simplicity while achieving the desired asymptotic bounds: constant-time query together with $O(n)$ space and preprocessing time. If, for some practical reason, the constant in the $O(n)$ space bound is of importance, one can look for improvements, which are not hard to find. We list two simple constant-factor improvements.

(1) The size $f(i)$ of B_i can be defined to be $2 \cdot 2^{\mathrm{rnz}(i)} + 1$ instead of $3 \cdot 2^{\mathrm{rnz}(i)}$. Moreover, assuming that all $a_i \geq 0$ (as is the case when using FS to solve Level Ancestors), we can use $\min(f(i), a_i)$. This eliminates B_0, and may give additional savings further on, depending on the shape of the tree in the Level Ancestor problem.

(2) The size of the arrays E, L in the reduction of Level Ancestors to the Find-Smaller problem can be cut in half by listing a vertex v in E only when visited by the Euler Tour for the last time (put otherwise, we list the vertices in post-order). It is still true that the level-l ancestor of v is the first vertex u occurring after v such that $\mathrm{level}(u) \leq l$. Thus, the reduction to Find Smaller is still correct. However, now the FS problem that results does not enjoy the (± 1) property. But it has a similar property: for all i, $a_{i+1} \geq a_i + 1$. Interestingly, this suffices for implementing the algorithm, at least with the micro-structure of Section 5.2. Thus, this saving in memory incurs no loss in running time.

Remark. This part of the solution is where the simplification with respect to [7] is most significant, although the outline (initial, non-optimal, solution, and usage of micro-blocks) is similar.

6 The Level-Descendant and Level-Successor Queries

Observation 1 can easily be turned from ancestors to descendants:

Observation 7 *Let $l > \mathrm{level}(v)$. Vertex u is the first level-l descendant of vertex v if and only if u is the first vertex after the first occurrence of v in the Euler tour such that*

level$(u) \geq l$*, provided that this vertex is a descendant of* v*. If it is not,* v *has no level-*l *descendant.*

By this observation, the level descendant query reduces to a Find-Greater problem, analogous to Find-Smaller and solved in the same way, plus a test of descendance. Thus, to add this functionality, we use the same arrays E, L and add a vector F maintaining the first occurrence of each vertex in the tour. We also need a search structure for "Find Greater." This structure is, of course, completely symmetric to the Find-Smaller structure so no further explanation should be necessary (incidentally, the micro table à-la Berkman-Vishkin can be shared). Testing for descendance is easy—u descends from v if and only if $F[v] \leq F[u] \leq R[v]$.

The level successor query is handled similarly, by the following observation:

Observation 8 *Vertex* u *is the level successor of vertex* v *if and only if* u *is the first vertex after the last occurrence of* v *in the Euler tour such that* level$(u) \geq$ level(v)*.*

7 Conclusion

I described how to construct and query a data structure for answering Level Ancestor queries on trees. The algorithm is based on Berkman and Vishkin's Euler Tour technique and is, in essence, a simplification of their PRAM algorithm. In contrast to the original, this version of the algorithm is simple and practical. The algorithm was implemented in C by Victor Buchnik; the code can be obtained from Amir Ben-Amram.

Another advantage of this algorithm is that it can be easily extended to support queries for Level Descendants and Level Successors.

References

[1] N. Alon and B. Schieber. Optimal preprocessing for answering on-line product queries. Technical report, Tel Aviv University, 1987.

[2] S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe. Nearest common ancestors: A survey and a new algorithm for a distributed environment. *Theory of Computing Systems*, 37(3):441–456, 2004.

[3] S. Alstrup and J. Holm. Improved algorithms for finding level ancestors in dynamic trees. In U. Montanari, J. D. P. Rolim, and E. Welzl, editors, *Proceedings of the 27th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 1853 of *LNCS*, pages 73–84. Springer-Verlag, July 2000.

[4] P. Beame and F. E. Fich. Optimal bounds for the predecessor problem and related problems. *J. Comput. Syst. Sci*, 65(1):38–72, 2002.

[5] M. A. Bender and M. Farach-Colton. The level ancestor problem simplified. *Theor. Comput. Sci*, 321(1):5–12, 2004.

[6] O. Berkman and U. Vishkin. Recursive *-tree parallel data-structure. In *30th Annual Symposium on Foundations of Computer Science*, pages 196–202. IEEE, 1989.

[7] O. Berkman and U. Vishkin. Finding level-ancestors in trees. *J. Computer and System Sciences*, 48(2):214–230, 1994.

[8] P. F. Dietz. Finding level-ancestors in dynamic trees. In *Workshop on Algorithms and Data Structures (WADS)*, pages 32–40, 1991.

[9] J. Fischer and V. Heun. A new succinct representation of rmq-information and improvements in the enhanced suffix array. In *Proceedings of the International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies (ESCAPE'07)*, volume ?? of *Lecture Notes in Computer Science*, pages ??–?? Springer, 2007.

[10] Gabow, Harold N. and Tarjan, Robert E. A linear time algorithm for a special case of disjoint set union. *J. Comput. Syst. Sci.*, 30:209–221, 1985.

[11] R. F. Geary, R. Raman, and V. Raman. Succinct ordinal trees with level-ancestor queries. In J. I. Munro, editor, *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004, New Orleans, Louisiana, USA, January 11-14, 2004*, pages 1–10. SIAM, 2004.

[12] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, May 1984.

[13] J. I. Munro and S. S. Rao. Succinct representations of functions. In J. Díaz, J. Karhumäki, A. Lepistö, and D. Sannella, editors, *Automata, Languages and Programming: 31st International Colloquium, ICALP 2004, Turku, Finland, July 12-16, 2004. Proceedings*, volume 3142 of *Lecture Notes in Computer Science*, pages 1006–1015. Springer, 2004.

[14] H. Yuan and M. J. Atallah. Efficient distributed third-party data authentication for tree hierarchies. In *28th IEEE International Conference on Distributed Computing Systems (ICDCS '08)*, pages 184–193. IEEE Computer Society, 2008.

[15] H. Yuan and M. J. Atallah. Efficient data structures for range-aggregate queries on trees. In R. Fagin, editor, *Database Theory—ICDT 2009, 12th International Conference, St. Petersburg, Russia, Proceedings*, volume 361 of *ACM International Conference Proceeding Series*, pages 111–120. ACM, 2009.